

ADANA SCIENCE AND TECHNOLOGY UNIVERSITY

Introduction to Computer Programming II

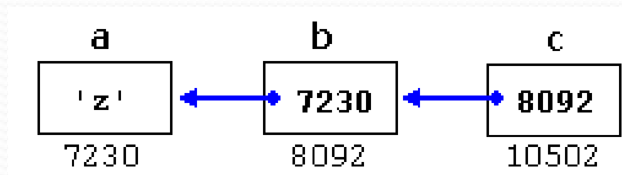
Pointers

to pointers

Pointers to pointers

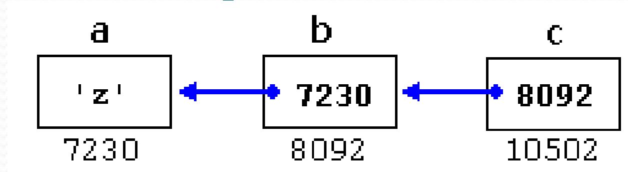
- C++ allows the use of pointers that <u>point to pointers</u>, that these, in its turn, point to data (or even to other pointers).
- The syntax simply requires an asterisk (*) <u>for each level</u> of indirection in the declaration of the pointer:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



- This, assuming the randomly chosen memory locations for each variable of 7230, 8092, and 10502, could be represented as:

Pointers to pointers



- With the value of each variable represented inside its corresponding cell, and their respective addresses in memory represented by the value under them.
- The new thing in this example is variable c, which is a pointer to a pointer, and can be used in three different levels of indirection, each one of them would correspond to a different value:
  - c is of type char** and a value of 8092
  - *c is of type char* and a value of 7230
  - **c is of type char and a value of 'z'

Pointers

void pointers

void Pointers

- The void type of pointer is a special type of pointer.
- In C++, void represents the *absence of type*.
- Therefore, void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).
- This gives void pointers a great flexibility, by being able to point to any data type, from an integer value or a float to a string of characters.
- In exchange, they have a great limitation: the data pointed to by them cannot be directly dereferenced (which is logical, since we have no type to dereference to),
  - and for that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

void Pointers

- One of its possible uses may be to pass generic parameters to a function.
- For example:

```
// increaser
#include <iostream>
using namespace std;

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data; ++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

```
y, 1603
```

void Pointers - important points

- All pointer types can be assigned to a pointer of type `void *` without casting.
- A `void *` pointer cannot be dereferenced.
  - The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer—for a pointer to `void`, this number of bytes cannot be determined from the type.

Invalid pointers and null pointers

- In principle, pointers are meant to point to valid addresses,
  - such as the address of a variable or the address of an element in an array.
- But pointers can actually point to any address, including addresses that do not refer to any valid element.
- Typical examples of this are *uninitialized pointers* and pointers to nonexistent elements of an array:

```
int * p;               // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

Invalid pointers and null pointers

```
int * p;               // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;  // element out of bounds
```

- Neither p nor q point to addresses known to contain a value, but none of the above statements causes an error.
- In C++, pointers are allowed to take any address value, no matter whether there actually is something at that address or not.
- What can cause an error is to dereference such a pointer (i.e., actually accessing the value they point to).
- Accessing such a pointer causes undefined behavior, ranging from an error during runtime to accessing some random value.

Invalid pointers and null pointers

- But, sometimes, a pointer really needs to explicitly point to nowhere, and not just an invalid address.
- For such cases, there exists a special value that any pointer type can take: the *null pointer value*.
- This value can be expressed in C++ in two ways: either with an integer value of zero, or with the **NULL** keyword:
  - `int * p = 0;`
  - `int * q = NULL;`
- Here, both p and q are *null pointers*,
  - meaning that they explicitly point to nowhere,
  - and they both actually compare equal: all *null pointers* compare equal to other *null pointers*.

Invalid pointers and null pointers

- NULL is defined in several headers of the standard library, and is defined as an alias of some *null pointer* constant value (such as 0).
- Do not confuse *null pointers* with void pointers!
- A *null pointer* is a value that any pointer can take to represent that it is pointing to "nowhere",
- while a void pointer is a type of pointer that can point to somewhere without a specific type.
- One refers to the value stored in the pointer, and the other to the type of data it points to.

DYNAMIC MEMORY

Dynamic memory

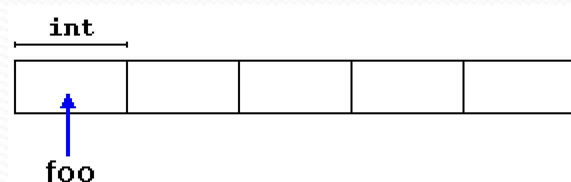
- Previously, we determined all memory needs before program execution by defining the variables needed.
- But there may be cases where the memory needs of a program can only be determined during runtime.
- For example, when the memory needed depends on user input.
- On these cases, programs need to ***dynamically allocate memory***, for which the C++ language integrates the operators ***new*** and ***delete***

Operators new and new[]

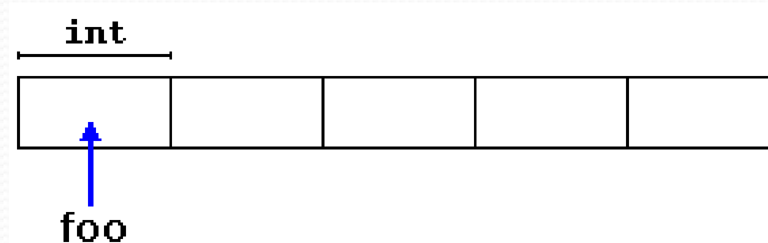
- Dynamic memory is allocated using operator ***new***.
- ***new*** is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.
- Its syntax is:
  - pointer = ***new*** type
  - pointer = ***new*** type [number_of_elements]

Operators new and new[]

- The first expression is used to allocate memory to contain one single element of type type.
- The second one is used to allocate a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these.
- For example:
  - int * foo;
  - foo = new int [5];
- In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer).
- Therefore, foo now points to a valid block of memory with space for five elements of type int.



Operators new and new[]



- Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expression foo[0] or the expression *foo (both are equivalent).
- The second element can be accessed either with foo[1] or *(foo+1), and so on...

Operators new and new[]

- There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using ***new***.
- The most important difference is that the size of a regular array needs to be a *constant expression*,
  - and thus its size has to be determined at the moment of designing the program, before it is run,
  - whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.

Operators new and new[]

- The dynamic memory requested by our program is allocated by the system from the memory heap.
- However, computer memory is a limited resource, and it can be exhausted.
- Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

Operators new and new[]

- C++ provides two standard mechanisms to check if the allocation was successful:
- One is by handling exceptions.
  - Using this method, an exception of type ***bad_alloc*** is thrown when the allocation fails.
  - Exceptions are a powerful C++ features
  - If this exception is thrown and it is not handled by a specific handler, the program execution is terminated.
- This exception method is the method used by default by new, and is the one used in a declaration like:
  - foo = new int [5]; // if allocation fails, an exception is thrown

Operators new and new[]

- The other method is known as ***nothrow***, and what happens when it is used is that
  - when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program,
  - the pointer returned by new is a *null pointer*, and the program continues its execution normally.
- This method can be specified by using a special object called ***nothrow***, declared in header <new>, as argument for new:
  - foo = new (nothrow) int [5];

Operators new and new[]

- foo = new (nothrow) int [5];
- In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

Operators delete and delete[]

- In most cases, memory allocated dynamically is only needed during specific periods of time within a program;
- once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory.
- This is the purpose of operator delete, whose syntax is:
  - delete pointer; delete[] pointer;
- The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([])

Operators delete and delete[]

- The value passed as argument to ***delete*** shall be
  - either a pointer to a memory block previously allocated with ***new***,
  - or a *null pointer* (in the case of a *null pointer* ***delete*** produces no effect).

Operators delete and delete[]

```
int main (){
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

How many numbers would you like to type? 5

Enter number : 75

Enter number : 436

Enter number : 1067

Enter number : 8

Enter number : 32

You have entered: 75, 436, 1067, 8, 32,

Operators new and delete

- Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant expression:
  - p= new (nothrow) int[i];
- There always exists the possibility that the user introduces a value for i so big that the system cannot allocate enough memory for it.
- For example, when I tried to give a value of 1 billion to the "How many numbers" question,
  - my system could not allocate that much memory for the program, and I got the text message we prepared for this case
    - (Error: memory could not be allocated).
- It is considered good practice for programs to always be able to handle failures to allocate memory, either by checking the pointer value (if nothrow) or by catching the proper exception.


Q&A